



[Back to Nylas Blogs](#)

The Nylas Engineering Blog



A New Search Parser

Improving IMAP Search in Nylas Mail

By: Mark Hahnenberg

May 8, 2017

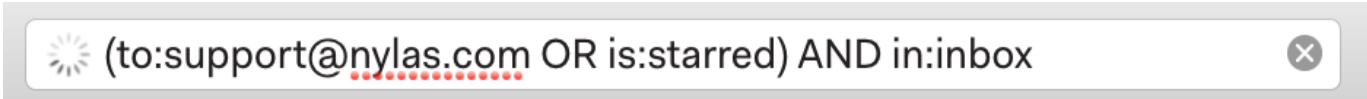
Creating a fast and relevant search experience is critical in sorting through all the emails we receive daily. Nylas Mail makes use of many technologies from both your mail provider (Gmail, Yahoo, etc) and our own search algorithms to find what you're looking for. While relevance was high with some of these mail providers, anyone using plain IMAP had a really poor search experience.

This post will focus on recent improvements to our search parser and query generator, the reasoning behind our choices, and possible future directions.

Some basic understanding of search engines and SQL is helpful.

Rethinking existing search

Modern email search has evolved considerably since the days of trying to find messages based on a single line in an email. These days, power users expect rich search terms that allow them to join, include, exclude, and sort all of their mail instantly. Queries like the following are not uncommon:

A screenshot of a search bar with a light gray background and a white input field. On the left of the input field is a sun-like icon. The text inside the input field is "(to:support@nylas.com OR is:starred) AND in:inbox". The email address "support@nylas.com" is underlined with red dots. On the right side of the input field is a small gray circle with a white "x" inside, indicating a clear button.

In order to support a search query like this we had to make some fundamental changes to our search systems:

- Tokenization and parsing of textual search query inputs into abstract syntax trees (ASTs). This had two benefits. First, it made the addition of new search features easier. Second, it made the transformation/lowering of search queries into other forms simpler.
- A new search query backend to generate local SQLite queries that utilize the FTS extension for fast prefix searching.
- A new search query backend to generate IMAP SEARCH commands that properly supports efficient scoped folder search (e.g. `in:inbox`).

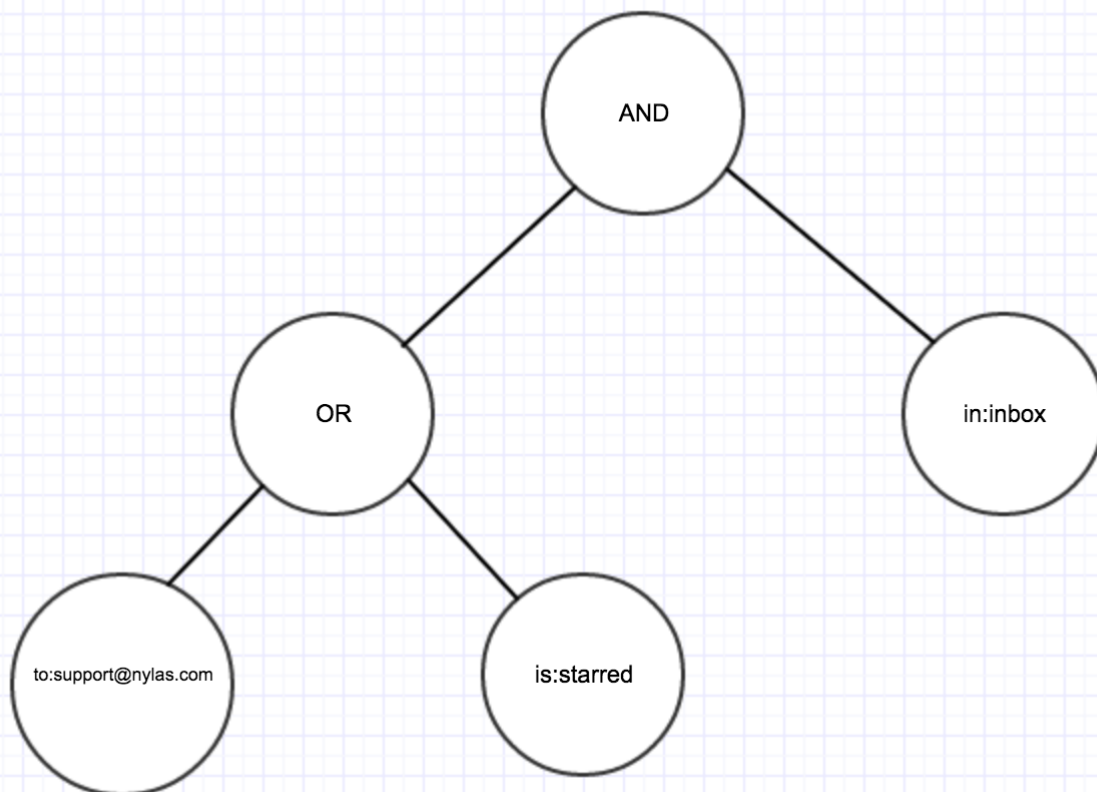
A new parser is born

Prior to our recent work on the search query engine in Nylas Mail, we had a very simple implementation of search that took the raw text typed by a user and stuffed it into primitive queries issued to our various backend providers. This often resulted in users being unable to find the messages or threads they were looking for. We knew we could do better.

The first step in improving search was to create a parser that would give us an extensible base to build on. In order to understand the parser let's start with the query we had above:

```
(to:support@nylas.com OR is:starred) AND in:inbox
```

This query has multiple terms, operators, and a strict precedence hierarchy. Our parser will take a recursive descent over the string to generate the following AST that then can be used to issue queries to our underlying search indices.



To create this we take a number of steps. You can follow along in the [nextToken](#) function.

First we tokenize the string into the distinct operator sub parts that we understand

```
* '('  
* 'to'  
* ':'
```

```
* support@nylas.com
* 'OR'
* 'is'
* ':'
* 'starred'
* ')'
* 'AND'
* 'in'
* ':'
* inbox
```

This stream of tokens is consumed by the SearchQueryParser. The first thing the parser sees is the '(' . This means that it is going to parse a subexpression and then hit a ')' , so it recurses and begins parsing again starting with 'to' . This is a keyword in our search syntax, so we look for the followup ':' and then the text after that. Taking all these together, we create a ToQueryExpression AST node. We then see the 'OR' keyword, so we parse the right hand side of the expression and create an OrQueryExpression . The parser continues on in this fashion until it has generated the AST referenced above.

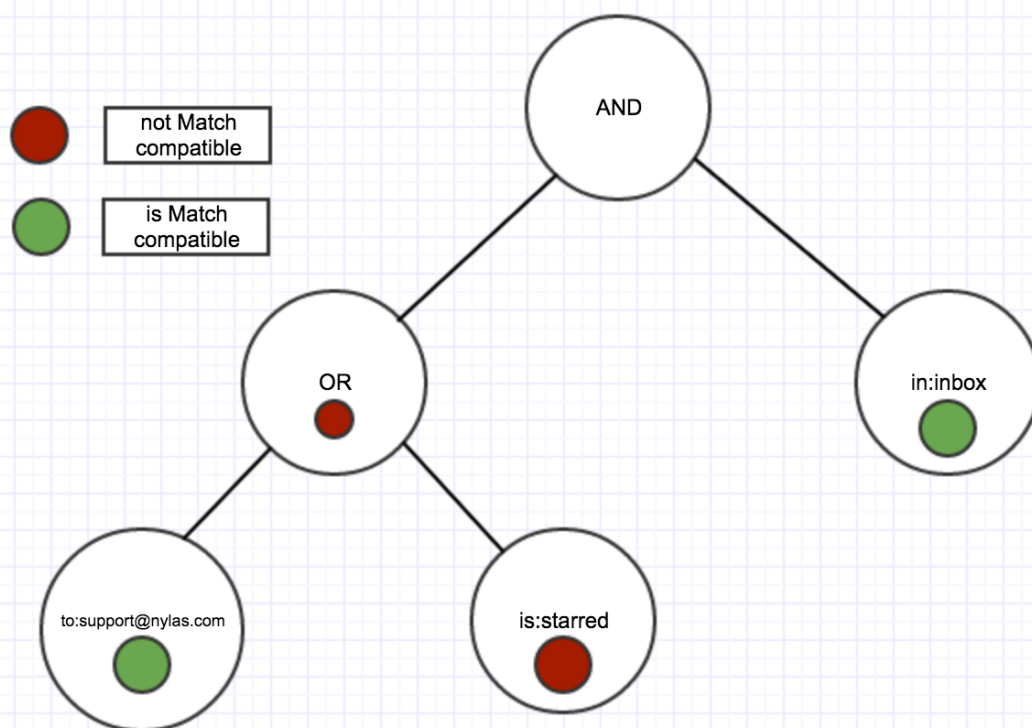
Converting the AST to full text search (FTS)

Nylas Mail maintains a local SQLite database of email data. To make searching fast, we issue SQL queries against this database so that any results that we have locally are returned instantly without having to consult a remote server. This also allows search to work without an internet connection.

Searching the database for the contents of emails using the SQLite LIKE operator could be very slow on large mailboxes, so we use SQLite's FTS5 extension to make prefix searches incredibly fast. However, we only store certain kinds of information in the FTS index like email bodies, subjects, and recipients. Other information, like whether an email is unread or starred, is stored in a normal SQLite table. This presents a small challenge when trying to generate queries, as FTS has its own syntax separate from that of SQLite which can only be accessed by using the **MATCH** operator.

To help us with this, we introduced the notion of **match compatibility** to the nodes of our AST. Any leaf node that can benefit from querying the FTS index is considered match compatible. For example, the node "to:support@nylas.com" would be considered match compatible because we store recipients in the FTS index, whereas the node "is:starred" would not be match compatible because it is stored directly in the SQL table. Additionally, internal nodes like "AND" and "OR" are only match compatible if both of their children are recursively match compatible.

We convert any node that is match compatible into query syntax for FTS5. Any match compatible nodes are converted to a new type of node called a MatchQueryExpression by a separate AST pass called the MatchCompatibleQueryCondenser. If an entire subtree is match compatible then all of its nodes are condensed into a single MatchQueryExpression node. Once this pass has been performed we can then easily generate a SQL query with the appropriate WHERE clause, composed of both standard SQLite expressions as well as MATCH expressions, which we then pass on to SQLite.

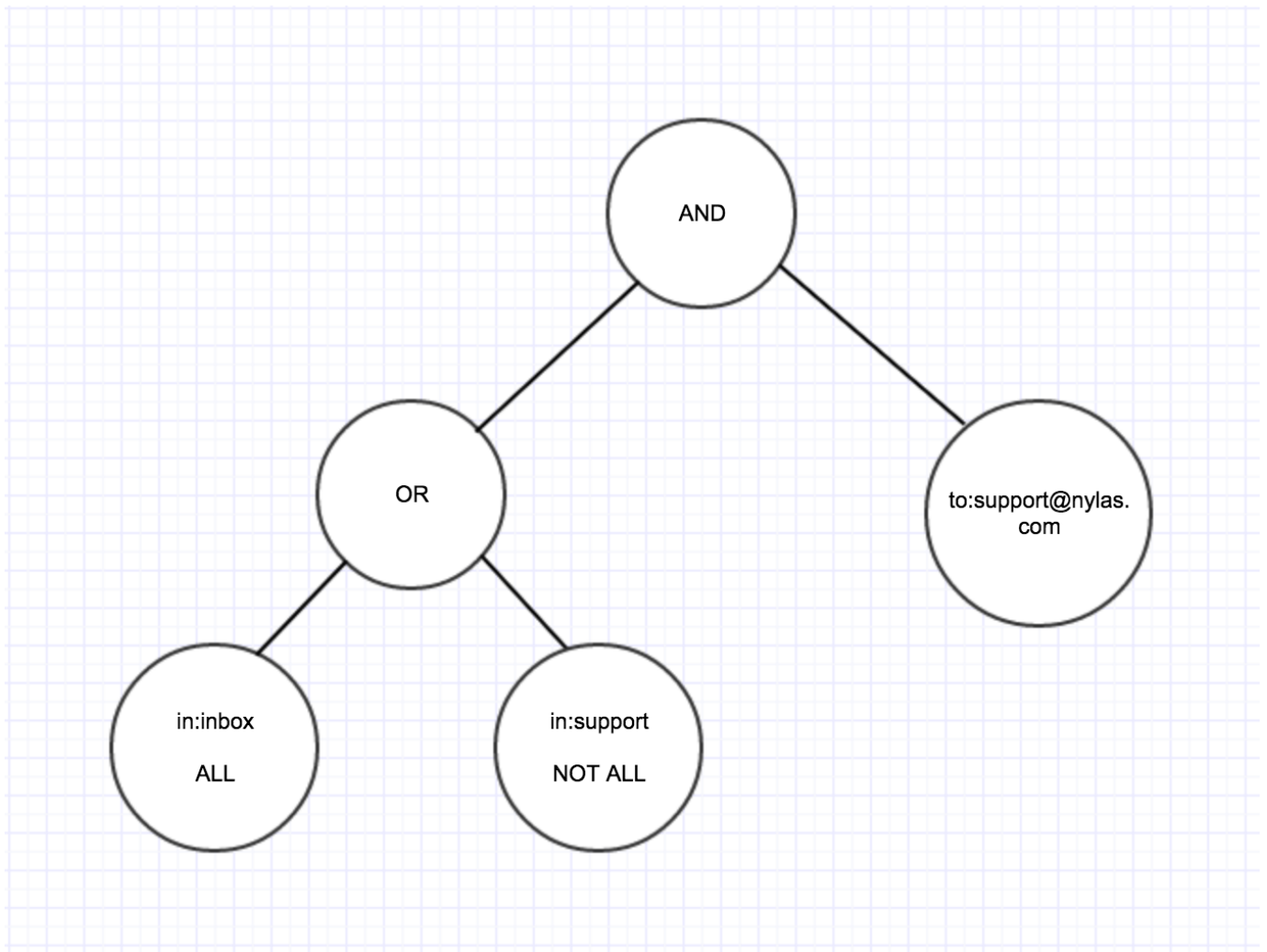


Supporting Folder Search for IMAP

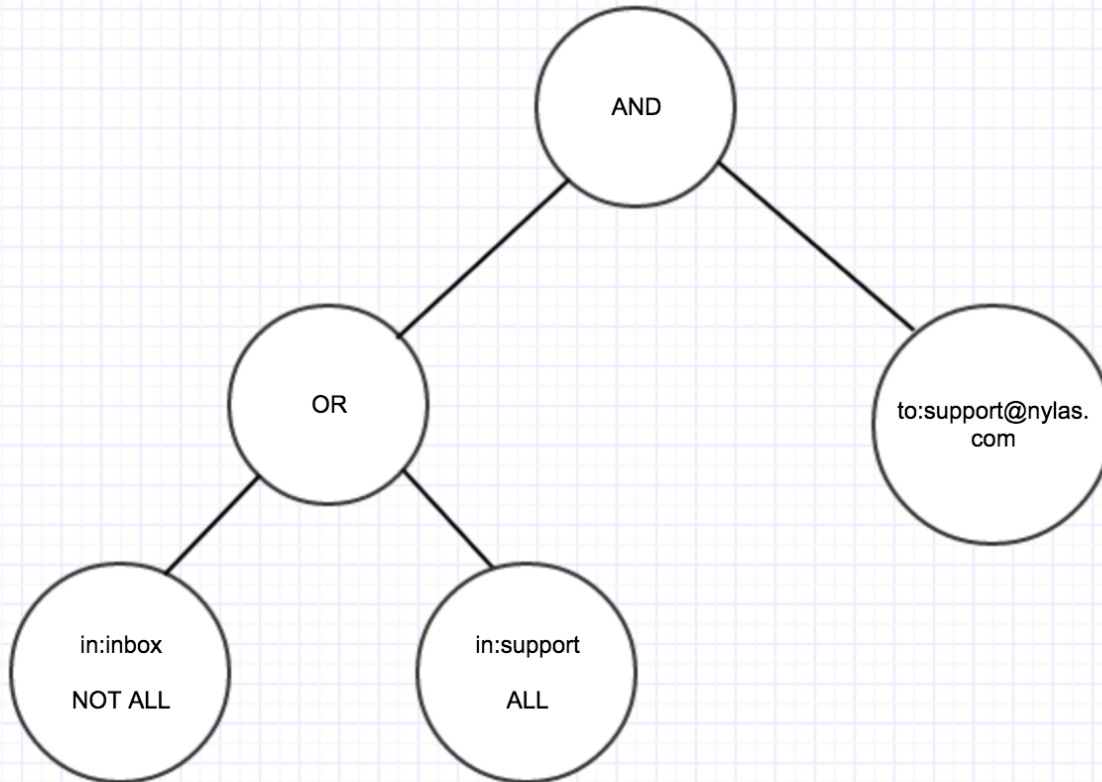
While local search is very fast, we don't always have the message you're looking for in the local database. This means we have to fall back to using the search facilities of the remote provider. In the case of IMAP, we can use the SEARCH command for clients to query the server for messages that fit a particular query. Clients construct queries from a variety of primitives including things like `TO <string>` , `SUBJECT <string>` , `BODY <string>` , and `FLAGGED` , among other things. It was pretty straightforward to translate our AST to this remote search query language with the exception of one feature: folder search with `in:foo` . The SEARCH command is scoped to an IMAP mailbox or folder. If we didn't support folder-scoped search we could simply issue the SEARCH command across all folders. However, the arbitrarily nested nature of our search query syntax prevents us from doing this. So how did we overcome this problem?

The key was in two of IMAP SEARCH's features: the `ALL` query and the `NOT <query>` . It's useful to think of SEARCH commands as set operations. So `ALL` represents the set of all messages in a particular folder. `NOT <query>` represents the set difference or complement operator. So the SEARCH command `ALL NOT SUBJECT foo` represents the set of all messages minus the set of messages that contain the subject `foo` . You can even obtain the empty set of messages with the command `NOT ALL` . This particular empty set query is the key to how we transform the search AST.

The IMAPSearchQueryExpressionVisitor class implements this transformation. The client code passes it a search AST and the folder to use as context. As the visitor processes the AST, if it comes to an `InQueryExpression` node it checks the node's folder against the current folder context. If they match, it replaces the node with an `ALL` command. If they don't match, it replaces the node with a `NOT ALL` command. So running this pass over a slightly modified example AST for the folder `inbox`, we would get the following result:



And if we ran it for another folder, say "Support", we would get the following result:



Voila! We now support folder-scoped queries for IMAP!

Optimizing Folder Search

Scoped folder search is all well and good, but there's a bit of an inefficiency here. We correctly generate the query for each folder, but we still have to run it on every folder! In a simple query like `in:inbox AND from:foo`, we actually only care about one folder. That's a lot of extra network interaction if the IMAP account has, say, 50 folders total! It would be nice if we could figure out which folders we need to search based on the query itself. It turns out that that is pretty easy with another simple pass over the AST.

The [IMAPSearchQueryFolderFinderVisitor](#) implements a simple set of rules that enable us to determine for which folders we should issue a particular query. The rules are as

follows:

- $\text{Folders}(\text{AND}(\text{child1}, \text{child2})) \Rightarrow \text{Folders}(\text{child1}) \cap \text{Folders}(\text{child2})$
- $\text{Folders}(\text{OR}(\text{child1}, \text{child2})) \Rightarrow \text{Folders}(\text{child1}) \cup \text{Folders}(\text{child2})$
- $\text{Folders}(\text{IN}(\text{foo})) \Rightarrow \{\text{foo}\}$
- $\text{Folders}(_) \Rightarrow \text{TOP}$

In other words, the folders for **AND** are the intersection of the folders of its children, the folders for **OR** are the union of its children, the folders for **IN** is the set containing the single folder referred to, and all other operations operate over all folders (which refer to by a special value called **TOP**). An interesting effect of this optimization is that for search queries like `in:foo AND in:bar`, we end up not searching any folders at all because it is impossible for a message to be in two folders at once!

There are further optimizations we can perform on the transformed AST such as simplifying nodes like **ALL AND NOT ALL** into **NOT ALL**, etc. This is left as an exercise for the reader.

Final Thoughts

Having really reliable, feature-rich search can easily make or break any data rich application. We've built a solid, extensible foundation for future expansion of Nylas Mail's search features. There are a number of ideas that we have about where to go next including adding negation support, implementing robust contact search using an n-gram powered index, measuring search quality, and more.

[Terms](#) · [Privacy](#) · [Copyright](#)

Follow us  

